Homework 3: Parametricity, PCF and Recursive Types

15-814: Types and Programming Languages
Fall 2015
TA: Evan Cavallo (ecavallo@cs.cmu.edu)

Out: 10/8/15
Due: 10/20/14, 10:30am

1 Parametricity

In this section, we will prove some metatheoretic results about System **F** extended with `nat` using parametricity. To review, we can think of the parametricity theorem as stating that two different notions of equality of expressions coincide:

Proposition 1 (PFPL Corollary 48.17) *$e \cong e'$ if and only if $e \sim_\tau e'$.*

The first of these equalities, $\cong$, is *observational equivalence*. Without going into details, two expressions are observationally equivalent if they behave the same way in any context. A bit more precisely, they are equivalent if one can be substituted for the other in any closed expression of type `nat` without changing its result. (The choice of `nat` is somewhat arbitrary – the key is that there are at least two distinguishable values of type `nat`.)

The second equality, $\sim_\tau$, is *logical equivalence*. Unlike observational equivalence, the definition of the $e \sim_\tau e'$ depends on τ in an essential way. This is another example of a *logical relation*, like the reducibility predicate we considered in Homework 2. Indeed, parts of the definition are very similar; for example, by definition,

> $e \sim_{\tau_1 \to \tau_2} e'$ holds iff for any e_1, e_1', $e_1 \sim_{\tau_1} e_1'$ imples $e(e_1) \sim_{\tau_2} e'(e_1')$.

As with the reducibility predicate, the definition of $\sim_{\tau_1 \to \tau_2}$ depends inductively on the definitions of $\sim_{\tau_1}$ and $\sim_{\tau_2}$. This is perfectly fine, since τ_1 and τ_2 are structurally smaller types than $\tau_1 \to \tau_2$. However, the situation becomes more complicated when we introduce polymorphism. We might hope to define logical equivalence at a polymorphic type by

> (?) $e \sim_{\forall t.\tau} e'$ holds iff, for any type ρ, $e[\rho] \sim_{[\rho/t]\tau} e'[\rho]$ holds.

Unlike the previous definition, however, this is not well-defined, because this definition of $\sim_{\forall t.\tau}$ references the definition of $\sim_{[\rho/t]\tau}$ for *every* type ρ. In turn, $\sim_{[\rho/t]\tau}$ may depend on the definition of $\sim_\rho$. Thus, the meaning of $\sim_{\forall t.\tau}$ may depend on the meaning of $\sim_\rho$ for every type ρ! We want to avoid this kind of circularity. (For those interested in looking further into this issue, it stems from the *impredicativity* of polymorphic types.) The solution, roughly speaking, is to avoid counting on a previously defined $\sim_\rho$ by instead quantifying over *all relations* (satisfying some requirements). In other words, we define

> $e \sim_{\forall t.\tau} e'$ holds iff, for any choice of admissible relation $R : \rho \leftrightarrow \rho'$, $e[\rho] \sim_\tau e'[\rho']$ holds when we treat R as the definition of $e \sim_t e'$.

Here, $R : \rho \leftrightarrow \rho'$ means that R is a relation between expressions of type ρ and expressions of type ρ'. An *admissible* relation $R : \rho \leftrightarrow \rho'$ is one for which the following conditions hold:

1. Respect for observational equivalence: If $R(e, e')$ and $d \cong_\rho e$ and $d' \cong_{\rho'} e'$, then $R(d, d')$.
2. Closure under converse evaluation: If $R(e, e')$, then if $d \mapsto e$, then $R(d, e')$ and if $d' \mapsto e'$, then $R(e, d')$.

(To be completely precise, $\sim$ is defined with respect to an environment mapping type variables to relations, and this definition says that to establish $e \sim_{\forall t.\tau}$ we must check that $e \sim_\tau e'$ holds for any extension of the environment by $t \hookrightarrow (R : \rho \leftrightarrow \rho')$ with $R : \rho \leftrightarrow \rho'$ admissible. See PFPL 48 for a thorough formal treatment.)

Although this definition of $e \sim_{\forall t.\tau} e'$ may seem overly strict, Proposition 1 tells us that it does in fact coincide with observational equivalence. Thus, we can prove results about observational equivalence, in a sense the most obvious notion of equivalence, by using the more manageable notion of logical equivalence. Before proving anything, though, here is the full definition of logical equivalence for our language:

1. $e \sim_{\tau_1 \to \tau_2} e'$ holds iff for any e_1, e_1', $e_1 \sim_{\tau_1} e_1'$ imples $e(e_1) \sim_{\tau_2} e'(e_1')$.
2. $e \sim_{\forall t.\tau} e'$ holds iff, for any choice of admissible relation $R : \rho \leftrightarrow \rho'$, $e[\rho] \sim_\tau e'[\rho']$ holds when we treat R as the definition of $e \sim_t e'$.
3. $e \sim_{\texttt{nat}} e'$ holds iff there is a number n such that $e \mapsto^* \overline{n}$ and $e' \mapsto^* \overline{n}$. (For sake of simplicity, we're assuming an eager dynamics for `nat`.)

One important consequence of Proposition 1 is the fact that logical equivalence is reflexive: for any expression $e : \tau$, $e \sim_\tau e$ holds. This is what PFPL calls the parametricity theorem. (Showing that it holds forms the bulk of the proof of Proposition 1.)

Task 1 *Show that for any expression e of type $\forall t.t \to t$ and any expression $e_0 : \rho$, the terms $e[\rho](e_0) : \rho$ and $e_0 : \rho$ are observationally equivalent (that is, $e[\rho](e_0) \cong e_0$). (Hint: You know that $e \sim_{\forall t.t \to t} e$. Also, observe that the relation $S : \rho \leftrightarrow \rho$ defined by*

$S(e_1, e_2)$ *holds iff* $e_1 \cong e_2 \cong e_0$

is admissible.)

Task 2 *Using the previous task, show that any expression e of type $\forall t.t \to t$ is equivalent to the identity function. (Hint: Unroll the definition of $\forall t.t \to t$ and use the properties of admissible relations.)*

Task 3 *Show that for any expression e of type $\forall t.t \to t \to t$ and any $e_1 : \rho$ and $e_2 : \rho$, either $e[\rho](e_1)(e_2) \cong e_1$ or $e[\rho](e_1)(e_2) \cong e_2$.*

2 Defining Inductive and Coinductive Types

In this section, we will encode inductive and coinductive types, as seen on the last assignment, as recursive types in **FPC**. We will be working in **FPC** extended with general recursion ($\texttt{rec}(t.\tau)$ can be encoded using recursive types, so this is just for convenience).

$$\begin{array}{rcl} \tau & ::= & t \mid \tau \rightharpoonup \tau \mid \tau \times \tau \mid \tau + \tau \mid \texttt{rec}(t.\tau) \\ e & ::= & \cdots \mid \texttt{fold}\{t.\tau\}(e) \mid \texttt{unfold}\{t.\tau\}(e) \mid \texttt{fix}\{\tau\}(x.e) \end{array}$$

2.1 Inductive Types

For now we will work in eager **FPC**, where $\texttt{fold}\{t.\tau\}(e)$ is only a value when e is a value. It shouldn't be too hard to convince yourself that the following encodings are correct:

$$\mu t.\tau' \triangleq \texttt{rec}(t.\tau')$$
$$\texttt{fold}\{t.\tau'\}(e) \triangleq \texttt{fold}\{t.\tau'\}(e)$$

Task 4 *[5] Encode the* **M** *expression* $\texttt{rec}\{t.\tau'\}(x.e_1; e_2)$ *in the extension of* **FPC** *described above. For convenience, abbreviate* $\tau \triangleq \texttt{rec}(t.\tau')$.

(Hint) *You will want to use generic programming. It may help to think about how to do this for* `nat` *and then generalize.*

2.2 Laziness and Streams

In defining coinductive types using the same strategy as above, things get a bit tricky. Let's consider the special case of streams for now.

$$\texttt{stream} \triangleq \nu t.\texttt{nat} \times t$$

Suppose we define the stream type in **FPC** as follows:

$$\texttt{stream} \triangleq \texttt{rec}(t.\texttt{nat} \times t)$$
$$\texttt{hd}(e) \triangleq \texttt{fst}\ \texttt{unfold}\{t.\texttt{nat} \times t\}(e)$$
$$\texttt{tl}(e) \triangleq \texttt{snd}\ \texttt{unfold}\{t.\texttt{nat} \times t\}(e)$$

Unfortunately, we will run into problems when we try to get our hands on a value of this type (by constructing it using either general recursion or something like `strgen`.) Such a value must be of the form

$$\texttt{fold}\{t.\texttt{nat} \times t\}(\langle v_0, \texttt{fold}\{t.\texttt{nat} \times t\}(\langle v_1, \ldots\rangle)\rangle)$$

But the expression elided by "..." must be infinite, and so this will never evaluate to a value. This is not a problem in the lazy variant of **FPC**, where the dynamics contain the following rule:

$$\frac{}{\texttt{fold}\{t.\tau\}(e)\ \mathsf{val}}$$

In this setting, $\texttt{fold}\{t.\tau\}(e)$ is a value for any e and so the second component of the product (the tail of the stream) will not be computed until it is needed.

Task 5 *[3] We could add laziness to* **FPC** *in other ways. Consider taking the eager variant and replacing the value rule for pairs with this rule:*

$$\frac{}{\langle e_1, e_2\rangle\ \mathsf{val}}$$

In this setting, `fold` *isn't lazy, but pairs are; all pairs are values.*

1. *Does the above encoding of streams work with this dynamics? If so, give an example of a canonical form of stream type. If not, explain why.*
2. *Same question, but imagine pairing is only lazy in the second component, i.e.*

$$\frac{v_1\ \mathsf{val}}{\langle v_1, e_2\rangle\ \mathsf{val}}$$

3. *Same question, where pairing is only lazy in the first component.*

We will say one last thing about laziness before moving on with our encoding of streams and general coinductive types. You may well ask at this point how laziness affects our encoding of inductive types.

Task 6 *[3] Consider the type* $\texttt{rec}(t.\texttt{unit} + t)$*, which would be our encoding of* `nat` *in* **FPC**. *Prove the following statement or give a counterexample.*

Proposition 2 *In* the lazy variant *of* **FPC** *(the version where* `fold` *is lazy), every closed value of type* $\texttt{rec}(t.\texttt{unit} + t)$ *is extensionally equivalent to (a suitable encoding of)* $\overline{n}$ *for some* n.

You can use the constructors `z` *and* `s` *without defining their encodings. Your reasoning can be fairly informal, but should be convincing.*

2.3 Simulating Laziness

Even if we are working in the eager variant of **FPC**, we can simulate lazy evaluation using a type of *suspended computations* or *suspensions*.

$$\begin{array}{rcl} \tau & ::= & \cdots \mid \texttt{susp}(\tau) \\ e & ::= & \cdots \mid \texttt{susp}(e) \mid \texttt{force}(e) \mid \texttt{fix}(x.e) \end{array}$$

$$\frac{\Gamma \vdash e : \mathtt{unit} \rightharpoonup \tau}{\Gamma \vdash \mathtt{susp}(e) : \mathtt{susp}(\tau)}\ (\textsc{Susp-I}) \quad \frac{\Gamma \vdash e : \mathtt{susp}(\tau)}{\Gamma \vdash \mathtt{force}(e) : \tau}\ (\textsc{Susp-E}) \quad \frac{\Gamma, x : \mathtt{susp}(\tau) \vdash e : \mathtt{susp}(\tau)}{\Gamma \vdash \mathtt{fix}(x.e) : \mathtt{susp}(\tau)}\ (\textsc{Fix-Susp-S})$$

$$\frac{e\ \mathsf{val}}{\mathtt{susp}(e)\ \mathsf{val}} \quad \frac{e \mapsto e'}{\mathtt{susp}(e) \mapsto \mathtt{susp}(e')}\ (\textsc{Susp-Step}) \quad \frac{}{\mathtt{fix}(x.e) \mapsto [\mathtt{fix}(x.e)/x]e}\ (\textsc{Fix-Susp-D})$$

$$\frac{e \mapsto e'}{\mathtt{force}(e) \mapsto \mathtt{force}(e')}\ (\textsc{Force-Step}) \quad \frac{}{\mathtt{force}(\mathtt{susp}(\lambda x{:}\mathtt{unit}.e)) \mapsto [\langle\rangle/x]e}\ (\textsc{Susp-Force})$$

An expression of type τ is suspended by wrapping it in a $\mathtt{unit} \rightharpoonup \tau$ function and packaging this with a `susp`. Thus, the expression will not yet be reduced down to a value. A suspension may be resumed by calling `force`. When a suspension is forced, the underlying function body is evaluated (with $\langle\rangle$ substituted for the `unit` argument.) This technique is common for implementing lazy evaluation.

Task 7 *[5] Lest you forget how to do a type safety proof, show that progress and preservation remain true with the addition of suspensions. You need only consider the cases related to suspensions, i.e. the new rules above. There will be three cases for progress and four for preservation. You may assume that appropriately extended canonical forms and substitution lemmas hold.*

Suspensions can be encoded in eager languages (think about how to do this; it's straightforward.) From this perspective, eager evaluation is actually more expressive than lazy evaluation, because we can simulate lazy evaluation by suspensions.

Combining suspensions and recursive types, we can define streams as follows:

$$\mathtt{stream}_1 \triangleq \mathtt{rec}(t.\mathtt{nat} \times \mathtt{susp}(t))$$
$$\mathtt{hd}(e) \triangleq \mathtt{fst}\ \mathtt{unfold}(e)$$
$$\mathtt{tl}(e) \triangleq \mathtt{force}(\mathtt{snd}\ \mathtt{unfold}(e))$$
$$\mathtt{strgen}\ e\ \{\mathtt{hd}(x) \hookrightarrow e_1 \mid \mathtt{tl}(y) \hookrightarrow e_2\} \triangleq$$
$$(\mathtt{fix}\{\rho \rightharpoonup \mathtt{stream}_1\}(f.\lambda y{:}\rho.\mathtt{fold}(\langle [y/x]e_1, \mathtt{susp}(\lambda\_{:}\mathtt{unit}.f\ ([y/x]e_2))\rangle)))\ e$$

We will use _ in a binder when we don't use the argument, especially when it is of type `unit`.

Recursive types also allow us to define streams using general recursion instead of `strgen`:

$$\mathtt{ones} \triangleq \mathtt{fix}\{\mathtt{stream}_1\}(s.\langle \overline{1}, \mathtt{susp}(\lambda\_{:}\mathtt{unit}.s)\rangle)$$

Because the second component of the pair is suspended, it will not continue to evaluate when `fix` evaluates and `ones` is substituted for s, so the above expression will evaluate to a value in one step.

Task 8 *[4] Define an alternate encoding,* $\mathtt{stream}_2$ *(different from* $\mathtt{stream}_1$ *above), of the stream type in the eager variant of* **FPC**. *As in the above encoding, this will require using suspensions, but in a different part of the type. Define the operations* `hd`, `tl` *and* `strgen` *on your type.*

(Hint) *Adding laziness where we did corresponds to when we added lazy pairs (but not folds) to the language. What about the version where* `fold` *is lazy?*

Task 9 *[5]*

1. *Define an encoding of the coinductive type $\nu t.\tau'$ in* **FPC**. *You will need to use suspended computations.*
2. *Define* $\texttt{unfold}\{t.\tau'\}(e)$ *for your encoding of coinductive types.*
3. *Fill in the dots in the below implementation of* `gen`.

$$\texttt{gen}\{t.\tau'\}(x.e_1; e_2) \triangleq (\texttt{fix}\{\rho \rightharpoonup \tau\}(f.\lambda y{:}\rho.\texttt{fold}(\ldots \texttt{map}\{t.\tau'\}(x.fx; \ldots) \ldots))\ e_2$$

3 Halting Problem in PCF

Recall the language **PCF**:

$$\begin{array}{rcl} \tau & ::= & \texttt{nat} \mid \tau \rightharpoonup \tau \\ e & ::= & x \mid \texttt{z} \mid \texttt{s}(e) \mid \texttt{ifz}(e; e; x.e) \mid \lambda x{:}\tau.e \mid e\ e \mid \texttt{fix}\{\tau\}(x.e) \end{array}$$

Consider a term $H : (\texttt{nat} \rightharpoonup \texttt{nat}) \rightharpoonup \texttt{nat}$ with the following properties:

1. For all $f : \texttt{nat} \rightharpoonup \texttt{nat}$, either $H\ f \mapsto^* \texttt{z}$ or $H\ f \mapsto^* \texttt{s}(\texttt{z})$ (i.e. H always terminates and evaluates to either $\overline{0}$ or $\overline{1}$.)
2. $H\ f \mapsto^* \texttt{z}$ iff there exists n such that $f\ \texttt{z} \mapsto^* \overline{n}$ (i.e. $f\ \texttt{z}$ converges to a value.)
3. $H\ f \mapsto^* \texttt{s}(\texttt{z})$ iff $f\ \texttt{z}$ diverges.

Task 10 *[5] Prove that H is not definable in* **PCF**.

(Hint) *Suppose H exists. Define a term D (which may refer to H) such that D diverges iff $H\ D \mapsto^* \texttt{z}$ (to make a term diverge, you can easily write an infinite loop.) Then consider to what $H\ D$ should evaluate.*

This a version of the famous result that the Halting Problem is undecidable (in a sufficiently powerful language.) In the more general statement, H accepts a representation of the code of a function f instead of the function itself. This problem is also undecidable for **PCF**, but the proof is somewhat more complex. If you'd like to complete it for a bit of extra credit, continue on. Otherwise, you can stop here.

To make precise the idea of having H accept a representation of the code of a function, we use a technique called *Gödel-numbering*, which assigns a unique natural number to α-equivalence classes of terms. We will not go into details of how such a representation is computed, but you can see PFPL 9.4 for more information. We will write $\lceil e \rceil$ for the Gödel number of an expression e. Since natural numbers are available in **PCF**, this gives us a way of passing around representations of expressions as values that can be inspected arbitrarily (as opposed to functions themselves, which can only be "inspected" by application.)

We will also generalize the definition of H so that it accepts a natural number input as well as the function. The application $H\ \lceil f \rceil\ n$ evaluates to $\texttt{z}$ if $f\ n$ converges and evaluates to $\texttt{s}(z)$ if $f\ n$ diverges.

Bonus Task 1 *Prove that this more general H is not definable in* **PCF**.